rapids团队 - 香港科技大学 - 竞赛相关材料

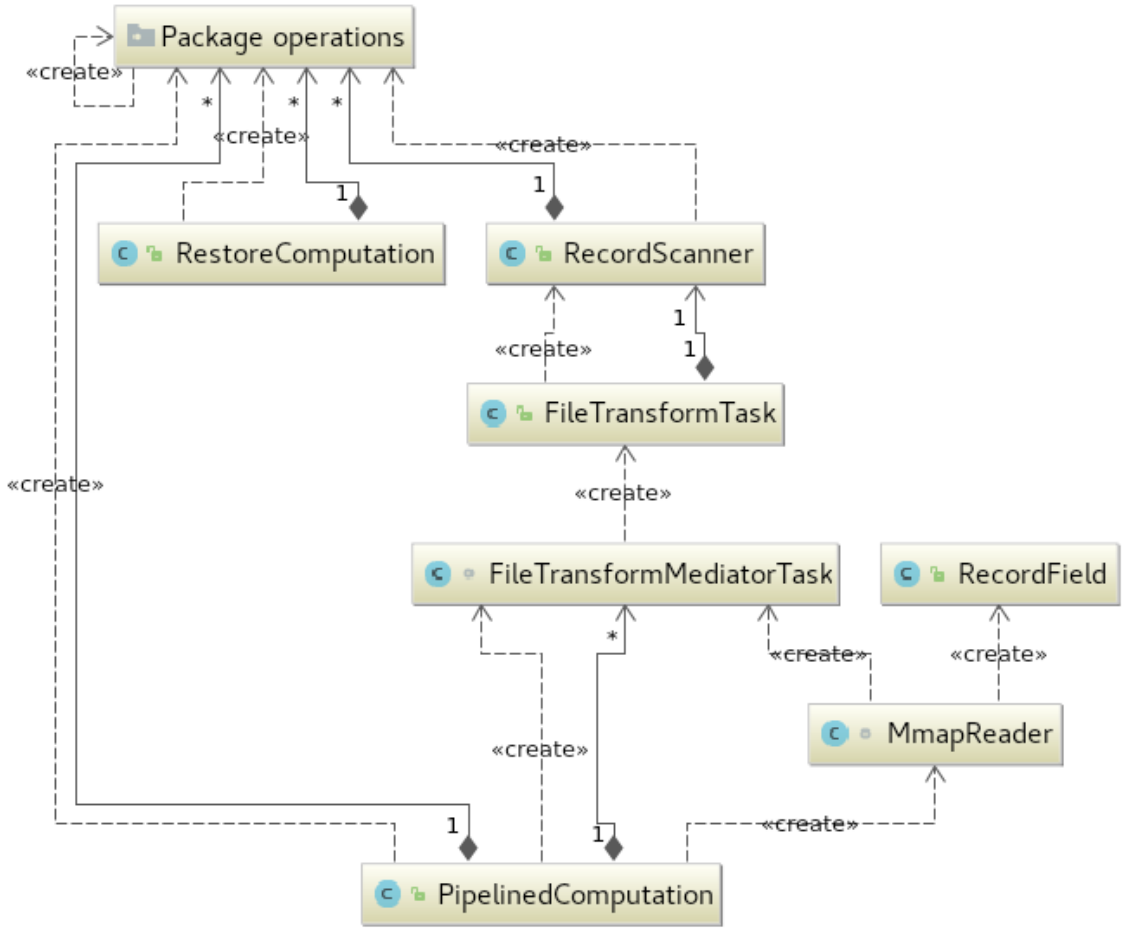
1. 算法设计思路和处理流程

1.1 基本思路

- 重放算法分为两个阶段，第一个阶段：单线程顺序读取十个文件，重放出数据库中最后时候符合主键在查询范围内的记录；第二个阶段：遍历第一阶段的记录数组，针对每一条记录，插入主键为key并且`byte[]`为value的`ConcurrentSkipListMap`中， 为之后产生出对应的文件作准备。
- Server段在程序启动时候，开启一个线程监听Client连接请求；在最后执行完第二阶段计算时候，遍历有序的 `ConcurrentSkipListMap`， 产生出结果文件对应的 `byte[]`，并使用 java nio 的 `transferFrom` 方式直接发送到Client并通过Client Direct Memory进行落盘。

1.2 第一阶段流水线的设计

整个重放算法有关的类都放在 `server2` 文件夹下， 其中的类关系如下图所示(通过jetbrains intellij生成)。



图中有四种不同的actor，这些actors的交互构成了完整的第一阶段计算的流水线：

- **actor 1: MmapReader(主线程)**， 负责顺序读取十个文件，按64MB为单位读取，若文件尾部不满64M就读取相应的大小, 读取之后对应的 `MappedByteBuffer` 会传入一个大小为1的 `BlockingQueue<FileTransformMediatorTask>`, 来让Mediator进行消费。因为阻塞队列的大小为1， 所以内存中最多只有三份 `MappedByteBuffer` (分别于主线程/Mediator线程/BlockingQueue中)， 总大小至多为192MB。

在获取下一块文件Chunk的时候，该Reader会判断是否已经初始化了关于单表的Meta信息。详细代码可见:
(其中RecordField类的类静态变量将用来记录这些Meta信息)。

```
// 1st work
private void fetchNextMmapChunk() throws IOException {
    int currChunkLength = nextIndex != maxIndex ? CHUNK_SIZE : lastCh
unkLength;

    MappedByteBuffer mappedByteBuffer = fileChannel.map(FileChannel.M
apMode.READ_ONLY, nextIndex * CHUNK_SIZE, currChunkLength);
    mappedByteBuffer.load();
    if (!RecordField.isInit()) {
        new RecordField(mappedByteBuffer).initFieldIndexMap();
    }

    try {
        mediatorTasks.put(new FileTransformMediatorTask(mappedByteBuf
fer, currChunkLength));
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

- **actor 2: Mediator(单个Mediator线程)**， 负责轮询 `BlockingQueue<FileTransformMediatorTask>` 来获取任务， 一个任务中包含一个 `MappedByteBuffer` 和对应的Chunk大小。

轮询的逻辑如下代码所示：

```
mediatorPool.execute(new Runnable() {
    @Override
    public void run() {
        while (true) {
            try {
                FileTransformMediatorTask fileTransformMediatorTask =
 mediatorTasks.take();
                if (fileTransformMediatorTask.isFinished)
                    break;
                fileTransformMediatorTask.transform();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
});
```

在最后读取完所有文件块的时候， 主线程会发送一个任务，通知 Mediator 可以结束了。该逻辑如下所示：

```
try {
    mediatorTasks.put(new FileTransformMediatorTask());
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

在收到任务后，Mediator负责分配， 保证每个Tokenizer and Parser处理的都是完整的块， 也就是说， 开始的index在`|mysql...`的`|`上， 结束的index在`\n`的后一个上。在这一步中，需要由Mediator维护好Chunk中末尾'\n'之后的bytes， 这也是Mediator最关键的工作之一。

关于任务分配，Mediator通过sumbit的方式向Tokenizer and Parser对应线程池提交任务， 并获取`Future<?>`传入下一个FileTransFormTask， 因为重放计算要求保证顺序， 一个任务做完后放入计算队列之前需要等上一个任务结束， 以保证顺序重放的正确性。一开始`Future<?>`的类静态对象被初始化为`isDone = true`。这一步的依赖至关重要， 保证了Log重放时候的顺序性，并且在最后做完tokenizer和parser任务后再放入taskQueue的设计最大程度利用了CPU。Mediator在解耦和简化并行计算模型方面发挥了重要作用。也是这个简洁的流水线设计中必不可少的一环节。

任务分配相关的核心代码如下(其中关键点在于start, end index的计算和prevRemainingBytes的维护以及prevFuture的维护):

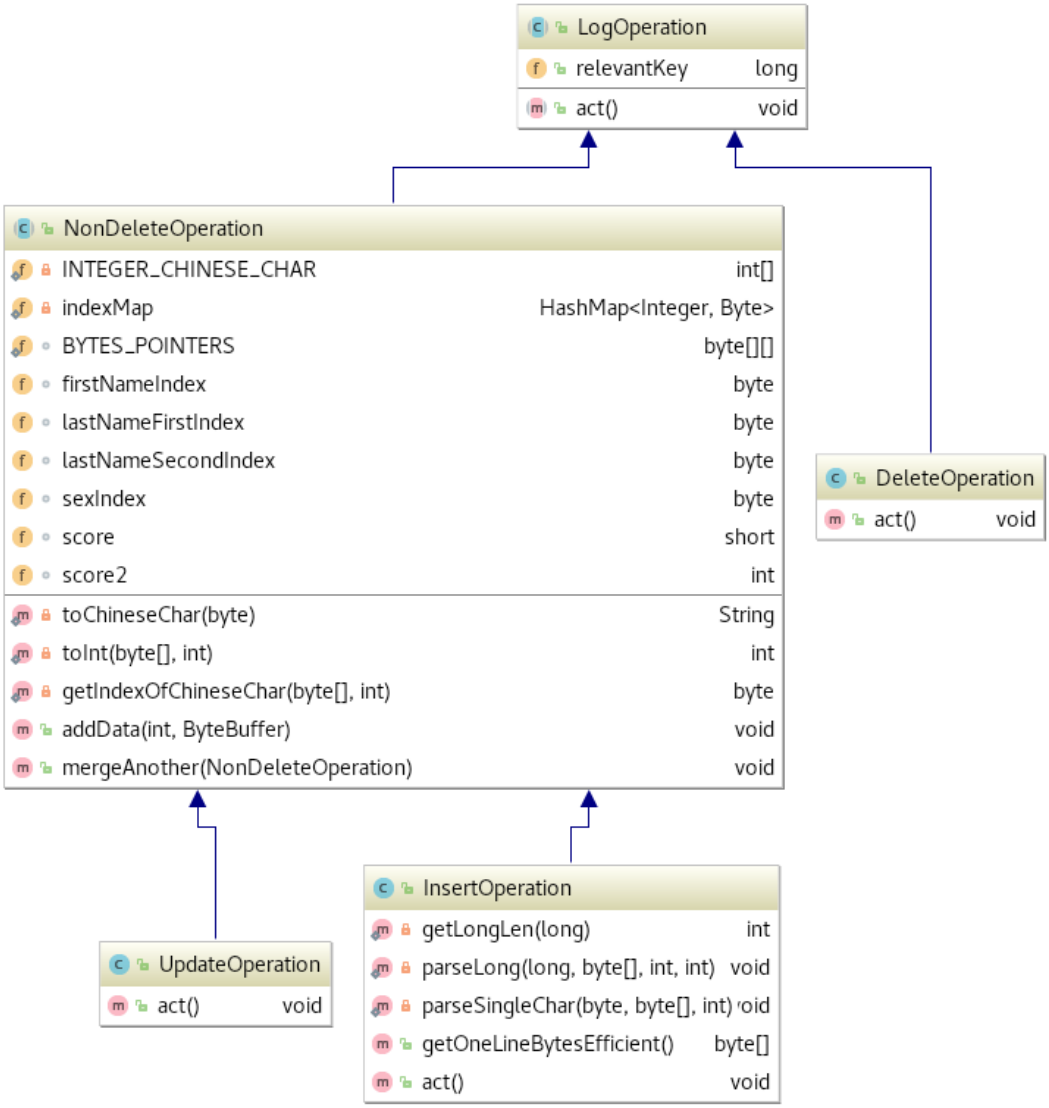
下面代码中的`submitIfPossible(FileTransformTask fileTransformTask)`方法中`serverPCGlobalStatus[globalIndex]`是根据统计出来的有用的Chunk，这是看其他队伍实现了5s以内的版本，不得以想到的，因为理论分析只有这样作或者利用其他数据的特征才可能实现5s以内的版本。这个一个取巧之处。这个取巧之处才可能帮助很多队伍创造出5s以内的时间，因为这样做的话，极限就是 ***1s的评测程序开销 + 2.5s mmap load 10G文件开销(完全和计算overlap,计算包括了tokenize, parse, restore) + 0.5s(并行eval，网络传输和落盘) = 4s***。

```
private static Future<?> prevFuture = new Future<Object>() {
    @Override
    public boolean cancel(boolean mayInterruptIfRunning) {
        return false;
    }

    @Override
    public boolean isCancelled() {
        return false;
    }

    @Override
    public boolean isDone() {
        return true;
    }

    @Override
    public Object get() throws InterruptedException, ExecutionExcepti
on {
        return null;
    }

    @Override
    public Object get(long timeout, TimeUnit unit) throws Interrupted
Exception, ExecutionException, TimeoutException {
        return null;
    }
};

private void submitIfPossible(FileTransformTask fileTransformTask) {
//        if (localPCGlobalStatus[globalIndex] == 1) {
    if (serverPCGlobalStatus[globalIndex] == 1) {
        prevFuture = fileTransformPool.submit(fileTransformTask);
        prevFutureQueue.add(prevFuture);
    }
    globalIndex++;
}

private void assignTransformTasks() {
    int avgTask = currChunkLength / WORK_NUM;

    // index pair
    int start;
    int end = preparePrevBytes();

    // 1st: first worker
    start = end;
    end = computeEnd(avgTask - 1);
```

```
        FileTransformTask fileTransformTask;
        if (prevRemainingBytes.limit() > 0) {
            ByteBuffer tmp = ByteBuffer.allocate(prevRemainingBytes.limit
    ());
            tmp.put(prevRemainingBytes);
            fileTransformTask = new FileTransformTask(mappedByteBuffer, s
    tart, end, tmp, prevFuture);
        } else {
            fileTransformTask = new FileTransformTask(mappedByteBuffer, s
    tart, end, prevFuture);
        }

        submitIfPossible(fileTransformTask);

        // 2nd: subsequent workers
        for (int i = 1; i < WORK_NUM; i++) {
            start = end;
            int smallChunkLastIndex = i < WORK_NUM - 1 ? avgTask * (i +
    1) - 1 : currChunkLength - 1;
            end = computeEnd(smallChunkLastIndex);
            fileTransformTask = new FileTransformTask(mappedByteBuffer, s
    tart, end, prevFuture);

            submitIfPossible(fileTransformTask);
        }

        // current tail, reuse and then put
        prevRemainingBytes.clear();
        for (int i = end; i < currChunkLength; i++) {
            prevRemainingBytes.put(mappedByteBuffer.get(i));
        }
    }
```

- **actor 3: Tokenizer and Parser for LogOperation(线程数为16的线程池)**
  这个逻辑在`FileTransformTask`中，负责对分配到某区间ByteBuffer里面的bytes进行解析，产生出用于重放的LogOperation对象来。 其中主要涉及到主键的解析，类型的解析和必要时LogOperation对象的创建。每个`FileTransformTask`对应一个唯一的`RecordScanner`， `RecordScanner`中封装了解析LogOperation对象的内容。中间设计到了利用表Meta信息减少访问bytes的优化，例如Delete操作的所有field都可以跳过，这也是比较容易发现的一个优化点。
- **actor 4: Restore Computation Worker(单个重放计算线程)**， 负责轮询获取任务进行计算， 当遇到大小为0的数组时候退出。重放计算线程轮询和退出的方式与Mediator类似，这里就不再给出。

LogOperation的相关类继承关系如下图所示((通过jetbrains intellij生成)):



为了取巧使用array代替hashmap我们不得不发现一个重要的规律：***主键变更并不会带来原来主键的属性***，比如主键从1->3，那么主键1原来的属性一定会被全update或者3不在range范围中，那么update key的操作就可以简单变成两个操作，一个delete之前主键，另一个insert新的主键。 这样才使得我们只要keep在范围内的主键相关记录，比如只有1000000到8000000的key对应记录有用，不会出现2^63的key有用，所以才可以使用array。

如果不取巧，我们也参考Trove Hashmap实现了一个 efficient的 hashmap，另外通过另一个hashset来记录range范围内的记录有哪些，这个实现并且在其它地方都不取巧，我们可以获得8.9s的成绩。

重放中，为了更memory-efficient，我们使用数组来模拟Hashmap表示对应的数据库，下标对应key, 引用对应value， 基于Range固定并且在int表示范围内

```
public static LogOperation[] ycheArr = new LogOperation[8 * 1024 * 10
24];
```

重放线程的逻辑就是顺序遍历取到的任务中每条LogOperation采取相应的行为。

```
static void compute(LogOperation[] logOperations) {
    for (LogOperation logOperation : logOperations) {
        logOperation.act();
    }
}
```

DeleteOperation的操作， 从数据库中删除记录

```
@Override
public void act() {
    ycheArr[(int) (this.relevantKey)] = null;
}
```

InsertionOperation的操作， 数据库中插入新的记录

```
@Override
public void act() {
    ycheArr[(int) (this.relevantKey)] = this;
}
```

Update操作， 从数据库中取出对应的记录，并进行属性更新

```
@Override
public void act() {
    InsertOperation insertOperation = (InsertOperation) RestoreComput
ation.ycheArr[(int) (this.relevantKey)]; //2
    if(insertOperation==null){
        insertOperation=new InsertOperation(this.relevantKey);
        RestoreComputation.ycheArr[(int) this.relevantKey]=insertOper
ation;
    }
    insertOperation.mergeAnother(this); //3
}
```

1.3 第二阶段Eval的并行执行

- 第二阶段的`byte[]` Evaluation是完全并行的，详细过程抽象出下面的代码, 其中`finalResultMap`为类型`public static final ConcurrentMap<Long, byte[]>`, 获取到的中间结果可以进一步被进行遍历生成最后有序的输出到文件的`byte[]`:

```
private static class EvalTask implements Runnable {
    int start;
    int end;
    LogOperation[] logOperations;

    EvalTask(int start, int end, LogOperation[] logOperations) {
        this.start = start;
        this.end = end;
        this.logOperations = logOperations;
    }

    @Override
    public void run() {
        for (int i = start; i < end; i++) {
            InsertOperation insertOperation = (InsertOperation) logOp
erations[i];
            if (insertOperation != null)
                finalResultMap.put(insertOperation.relevantKey, inser
tOperation.getOneLineBytesEfficient());
        }
    }
}

// used by master thread
static void parallelEvalAndSend(ExecutorService evalThreadPool) {
    LogOperation[] insertOperations = ycheArr;
    int lowerBound = (int) PipelinedComputation.pkLowerBound;
    int upperBound = (int) PipelinedComputation.pkUpperBound;
    int avgTask = (upperBound - lowerBound) / EVAL_WORKER_NUM;
    for (int i = lowerBound; i < upperBound; i += avgTask) {
        evalThreadPool.execute(new EvalTask(i, Math.min(i + avgTask,
 upperBound), insertOperations));
    }
}
```

- 其中 `insertOperation.getOneLineBytesEfficient()` 是一个优化点，如果使用StringBuilder实现会比较慢，我们的实现如下，避免使用StringBuild和调用append。在下面的代码中我们的实现主要使用了直接的 `byte[]` 的操作和自己写的转换 `parseLong` 和 `parseSingleChar`， 可以从原来基于StringBuild实现的 500ms cost减到 250ms。

```
private static int getLongLen(long pk) {
    int noOfDigit = 1;
    while ((pk = pk / 10) != 0)
        ++noOfDigit;
    return noOfDigit;
}

private static void parseLong(long pk, byte[] byteArr, int offset, in
t noDigits) {
    long leftLong = pk;
    for (int i = 0; i < noDigits; i++) {
        byteArr[offset + noDigits - i - 1] = (byte) (leftLong % 10 +
'0');
        leftLong /= 10;
    }
}

private static void parseSingleChar(byte index, byte[] byteArr, int o
ffset) {
    System.arraycopy(NonDeleteOperation.BYTES_POINTERS[index], 0, byt
eArr, offset, 3);
}

public byte[] getOneLineBytesEfficient() {
    byte[] tmpBytes = new byte[48];
    int nextOffset = 0;
    // 1st: pk
    int pkDigits = getLongLen(relevantKey);
    parseLong(relevantKey, tmpBytes, nextOffset, pkDigits);
    nextOffset += pkDigits;
    tmpBytes[nextOffset] = '\t';
    nextOffset += 1;

    // 2nd: first name
    parseSingleChar(firstNameIndex, tmpBytes, nextOffset);
    nextOffset += 3;
    tmpBytes[nextOffset] = '\t';
    nextOffset += 1;

    // 3rd: second name
    parseSingleChar(lastNameFirstIndex, tmpBytes, nextOffset);
    nextOffset += 3;
    if (lastNameSecondIndex != -1) {
        parseSingleChar(lastNameSecondIndex, tmpBytes, nextOffset);
        nextOffset += 3;
    }
    tmpBytes[nextOffset] = '\t';
    nextOffset += 1;
```

```
        // 4th: sex
        parseSingleChar(sexIndex, tmpBytes, nextOffset);
        nextOffset += 3;
        tmpBytes[nextOffset] = '\t';
        nextOffset += 1;

        // 5th score
        pkDigits = getLongLen(score);
        parseLong(score, tmpBytes, nextOffset, pkDigits);
        nextOffset += pkDigits;
        tmpBytes[nextOffset] = '\t';
        nextOffset += 1;

        // 6th score2
        if (score2 != -1) {
            pkDigits = getLongLen(score2);
            parseLong(score2, tmpBytes, nextOffset, pkDigits);
            nextOffset += pkDigits;
            tmpBytes[nextOffset] = '\t';
            nextOffset += 1;
        }
        tmpBytes[nextOffset - 1] = '\n';

        byte[] retBytes = new byte[nextOffset];
        System.arraycopy(tmpBytes, 0, retBytes, 0, nextOffset);
        return retBytes;
    }
```

- 第二阶段的后续处理可见代码， 生成出最后会落盘至文件的`byte[]`， 交给Server进行发送

```
public static void putThingsIntoByteBuffer(ByteBuffer byteBuffer) {
    for (byte[] bytes : finalResultMap.values()) {
        byteBuffer.put(bytes);
    }
}
```

1.4 网络传输和落盘：Server-Client 之间Zero-Copy

充分利用Direct Memory的特性，去除内核态和用户态拷贝。

- Client Side, API usage

```
FileChannel fileChannel = new RandomAccessFile(Constants.RESULT_HOME
 + File.separator + Constants.RESULT_FILE_NAME, "rw").getChannel();
nativeClient.start(fileChannel);
```

- 底层的实现, 使用了 `outputFile.transferFrom(clientChannel, 0, chunkSize);`， 直接从网络的clientChannel Zero-Copy到对应文件落盘， 不拷贝到用户态空间

```
public void start(FileChannel outputFile){
    if(outputFile == null){
        return;
    }
    try {
        clientChannel.write(ByteBuffer.wrap("A".getBytes()));
        int chunkSize = recvChunkSize();

        int recvCount = 0;
        ByteBuffer recvBuff = ByteBuffer.allocate(chunkSize);
        while (recvCount < chunkSize){
            recvCount += clientChannel.read(recvBuff);
        }
        String[] args = new ArgumentsPayloadBuilder(new String(recvBu
ff.array(), 0, chunkSize)).args;

        chunkSize = recvChunkSize();

        outputFile.transferFrom(clientChannel, 0, chunkSize);

        clientChannel.finishConnect();
        clientChannel.close();

    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

2. 创新点：算法设计上的创新点

2.1 流水线设计简洁， 耦合度低

流水线整体设计简洁，文件读取，mediator, tokenize/parser，计算之间耦合度低。并且有blockingQueue协调生产和消费之间关系。Mediator和计算线程都使用轮询的方式比submit task更为高效。

2.2 高效的Eval模块，网络传输 Zero-Copy， 并未使用第三方依赖

3. 健壮性

3.1 选手代码对不同的表结构适应性

现实生活中往往schema是确定的，要扩展到其他的表的时候，只需要修改 `NonDeleteOperation`。

1) 修改 field先, 这边的index对应到的是对应的char，如果char多的话把byte改为short即可

```
byte firstNameIndex = -1;
byte lastNameFirstIndex = -1;
byte lastNameSecondIndex = -1;
byte sexIndex = -1;
short score = -1;
int score2 = -1;
```

2) 修改 `addData` 和 `mergeAnother` 适应新的field

```
public void addData(int index, ByteBuffer byteBuffer) {
     switch (index) {
         case 0:
             firstNameIndex = getIndexOfChineseChar(byteBuffer.array
(), 0);
             break;
         case 1:
             lastNameFirstIndex = getIndexOfChineseChar(byteBuffer.ar
ray(), 0);
             if (byteBuffer.limit() == 6)
                 lastNameSecondIndex = getIndexOfChineseChar(byteBuff
er.array(), 3);
             break;
         case 2:
             sexIndex = getIndexOfChineseChar(byteBuffer.array(), 0);
             break;
         case 3:
             short result = 0;
             for (int i = 0; i < byteBuffer.limit(); i++)
                 result = (short) ((10 * result) + (byteBuffer.get(i)
 - '0'));
             score = result;
             break;
         case 4:
             int resultInt = 0;
             for (int i = 0; i < byteBuffer.limit(); i++)
                 resultInt = ((10 * resultInt) + (byteBuffer.get(i) -
 '0'));
             score2 = resultInt;
             break;
//             default:
//                 if (Server.logger != null)
//                     Server.logger.info("add data error");
//                 System.err.println("add data error");
     }
 }

 public void mergeAnother(NonDeleteOperation nonDeleteOperation) {
     if (nonDeleteOperation.score != -1) {
         this.score = nonDeleteOperation.score;
         return;
     }
     if (nonDeleteOperation.score2 != -1) {
         this.score2 = nonDeleteOperation.score2;
         return;
     }
     if (nonDeleteOperation.firstNameIndex != -1) {
         this.firstNameIndex = nonDeleteOperation.firstNameIndex;
```

```
            return;
        }
        if (nonDeleteOperation.lastNameFirstIndex != -1) {
            this.lastNameFirstIndex = nonDeleteOperation.lastNameFirstIn
dex;
            this.lastNameSecondIndex = nonDeleteOperation.lastNameSecond
Index;
            return;
        }
        if (nonDeleteOperation.sexIndex != -1) {
            this.sexIndex = nonDeleteOperation.sexIndex;
        }
    }
```

3.2 对不同DML变更的适应性(例如根据数据集特征过滤了变更数据，这些过滤操作是否能适应不同的变更数据集)。

8.9s的general实现可以适应任何的情况，并没有作任何针对复赛数据集的tricks。

4. 补充(8.9s general实现，即不利用数据集特征)

efficient hashmap for 8.9s 实现， 这个实现中不需要从hashmap中remove，因为我们使用了另一个hashset存range范围内的记录。对应的Restore逻辑如下代码:

4.1 数据和操作

- 两个关键的成员变量，一个记录数据库(含有垃圾，因为不remove,但包含数据库中当前所有信息和垃圾)，一个记录range范围内记录

```
public static YcheHashMap recordMap = new YcheHashMap(24 * 1024 * 102
4);
public static THashSet<LogOperation> inRangeRecordSet = new THashSet<
>(4 * 1024 * 1024);
```

- 对DeleteOperation 操作

```
@Override
public void act() {
    if (PipelinedComputation.isKeyInRange(this.relevantKey)) {
        inRangeRecordSet.remove(this);
    }
}
```

- 对InsertOperation 操作

```
@Override
public void act(){
    recordMap.put(this); //1
    if (PipelinedComputation.isKeyInRange(relevantKey)) {
        inRangeRecordSet.add(this);
    }
}
```

- 对UpdateOperation 操作

```
@Override
public void act(){
    InsertOperation insertOperation = (InsertOperation) recordMap.get
(this); //2
    insertOperation.mergeAnother(this); //3
};
```

- 对UpdateKeyOperation 操作

```
@Override
public void act() {
    InsertOperation insertOperation = (InsertOperation) recordMap.get
(this); //2
    if (PipelinedComputation.isKeyInRange(this.relevantKey)) {
        inRangeRecordSet.remove(this);
    }

    insertOperation.changePK(this.changedKey); //4
    recordMap.put(insertOperation); //5

    if (PipelinedComputation.isKeyInRange(insertOperation.relevantKe
y)) {
        inRangeRecordSet.add(insertOperation);
    }
}
```

4.2 HashMap probing detail 实现

- YcheLongHash.java

```
package com.alibaba.middleware.race.sync.server2;

import gnu.trove.impl.Constants;
import gnu.trove.impl.HashFunctions;

import java.util.Arrays;

/**
 * Created by yche on 6/24/17.
 */
public class YcheLongHash {
    /**
     * the set of longs
     */
    private transient long[] _set;

    /**
     * value that represents null
     * <p>
     * NOTE: should not be modified after the Hash is created, but is
     * not final because of Externalization
     */
    private long no_entry_value;

    /**
     * Creates a new <code>YcheLongHash</code> instance whose capacit
y
     * is the next highest prime above <tt>initialCapacity + 1</tt>
     * unless that value is already prime.
     *
     * @param initialCapacity an <code>int</code> value
     */
    YcheLongHash(int initialCapacity) {
        no_entry_value = Constants.DEFAULT_LONG_NO_ENTRY_VALUE;
        _set = new long[initialCapacity];
        //noinspection RedundantCast
        if (no_entry_value != (long) 0) {
            Arrays.fill(_set, no_entry_value);
        }
    }

    /**
     * initializes the hashtable to a prime capacity which is at leas
t
     * <tt>initialCapacity + 1</tt>.
     *
     * @param initialCapacity an <code>int</code> value
     * @return the actual capacity chosen
```

```
     */
    protected void setUp(int initialCapacity) {
        _set = new long[initialCapacity];
    }

    /**
     * Locates the index of <tt>val</tt>.
     *
     * @param val an <code>long</code> value
     * @return the index of <tt>val</tt> or -1 if it isn't in the se
t.
     */
    int index(long val) {
        int hash, index, length;

        length = _set.length;
        hash = HashFunctions.hash(val) & 0x7fffffff;
        index = hash % length;
        long state = _set[index];

        if (state == no_entry_value)
            return -1;

        else if (state == val)
            return index;

        return indexRehashed(val, index, hash, state);
    }

    private int indexRehashed(long key, int index, int hash, long sta
te) {
        // see Knuth, p. 529
        int length = _set.length;
        int probe = 1 + (hash % (length - 2));
        final int loopIndex = index;

        do {
            index -= probe;
            if (index < 0) {
                index += length;
            }
            //
            if (state == no_entry_value)
                return -1;

            //
            if (key == _set[index])
                return index;
        } while (index != loopIndex);
```

```
        return -1;
    }

    /**
     * Locates the index at which <tt>val</tt> can be inserted.  if
     * there is already a value equal()ing <tt>val</tt> in the set,
     * returns that value as a negative integer.
     *
     * @param val an <code>long</code> value
     * @return an <code>int</code> value
     */
    int insertKey(long val) {
        int hash, index;

        hash = HashFunctions.hash(val) & 0x7fffffff;
        index = hash % _set.length;
        long state = _set[index];


        if (state == no_entry_value) {
            insertKeyAt(index, val);

            return index;       // empty, all done
        } else if (_set[index] == val) {
            return -index - 1;   // already stored
        }

        // already FULL or REMOVED, must probe
        return insertKeyRehash(val, index, hash);
    }

    private int insertKeyRehash(long val, int index, int hash) {
        // compute the double hash
        final int length = _set.length;
        int probe = 1 + (hash % (length - 2));
        final int loopIndex = index;

        /**
         * Look until FREE slot or we start to loop
         */
        do {
            index -= probe;
            if (index < 0) {
                index += length;
            }
            long state = _set[index];

            // A FREE slot stops the search
```

```
            if (state == no_entry_value) {
                insertKeyAt(index, val);
                return index;
            }

            if (_set[index] == val) {
                return -index - 1;
            }

            // Detect loop
        } while (index != loopIndex);

        // We inspected all reachable slots and did not find a FREE o
ne
        // If we found a REMOVED slot we return the first one found

        // Can a resizing strategy be found that resizes the set?
        throw new IllegalStateException("No free or removed slots ava
ilable. Key set full?!!");
    }

    private void insertKeyAt(int index, long val) {
        _set[index] = val;  // insert value
    }
}
```

4.3 HashMap 接口 detail 实现

- YcheHashMap.java

```java
package com.alibaba.middleware.race.sync.server2;

import com.alibaba.middleware.race.sync.server2.operations.LogOperati
on;

/**
 * Created by yche on 6/23/17.
 */
public class YcheHashMap extends YcheLongHash {
    private transient LogOperation[] _values;

    YcheHashMap(int initialCapacity) {
        super(initialCapacity);
        setUp(initialCapacity);
    }

    public void setUp(int initialCapacity) {
        _values = new LogOperation[initialCapacity];
    }

    public LogOperation get(Object key) {
        int index = index(((LogOperation)key).relevantKey);
        return index < 0 ? null : _values[index];
    }

    private void doPut(LogOperation value, int index) {
        if (index < 0) {
            index = -index - 1;
        }
        _values[index] = value;
    }

    public void put(LogOperation key) {
        // insertKey() inserts the key if a slot if found and returns
 the index
        int index = insertKey(key.relevantKey);
        doPut(key, index);
    }
}
```